I217: Functional Programming

13. Verification of Arithmetic Calculator Compiler

Kazuhiro Ogata

Roadmap

- Correctness of Compiler
- Arithmetic Calculator
- Verification of Arithmetic Calculator Compiler

Correctness of Compiler

What is the correctness of a compiler C of a programming language L?

C takes a program p in L and generates a list of instructions that can be executed on a (virtual) machine M. Let $C(p)$ be the list of instructions and $M(C(p))$ be the result obtained by executing $C(p)$ on M.

One possible definition of the correctness of C: $M(C(p))$ is correct for all programs p.

How do we define that $M(C(p))$ is correct?

Correctness of Compiler

Another way to execute p: interpreting p based on the operational semantics of L. Let I be the interpreter of L and $I(p)$ be the result obtained by interpreting p with I.

We regard I as the oracle to verify the correctness of C.

The (relative) correctness of C with respect to I : for all programs p in L, $M(C(p)) = I(p)$.

We will verify that a compiler of an arithmetic calculator is correct.

Arithmetic Calculator

Natural numbers a la Peano in which the division-by-zero exception is taken into account are first sepcified:

```
mod! PNAT principal-sort PNat&Err {
 [PZero NzPNat < PNat]
 [PNat ErrPNat < PNat&Err]
 op 0 : -> PZero {constr} .
 op s : PNat -> NzPNat {constr} .
 op errPNat : -> ErrPNat {constr} .
 --
 vars M N : PNat .
 vars ME NE : PNat&Err .
```


Arithmetic Calculator

```
eq (0 = s(N)) = false .
eq (s(M) = s(N)) = (M = N) .
eq (errPNat = N) = false .
```

Note that for each sort *S* the following operator and equation are declared in advance:

```
op _=_ : S S -> Bool {comm} .
eq (X = X) = true .
```

where X is a variable of *S*.

```
op if_then{_}else{_} : Bool PNat&Err PNat&Err -> PNat&Err .
eq if true then {ME} else {NE} = ME .
eq if false then {ME} else {NE} = NE .
```

Arithmetic Calculator

op _<_ : PNat PNat -> Bool .
eq 0 < 0 = false .
eq 0 < s(N) = true .
eq s(M) < 0 = false .
eq s(M) < s(N) = M < N .

op _+_ : PNat PNat -> PNat .
op _+_ : PNat&Err PNat&Err -> PNat&Err .
eq 0 + N = N .
eq s(M) + N = s(M + N) .
eq ME + errPNat = errPNat .
eq errPNat + NE = errPNat .

Arithmetic Calculator

op _*_ : PNat PNat -> PNat .
op _*_ : PNat&Err PNat&Err -> PNat&Err .
eq 0 * N = 0 .
eq s(M) * N = N + (M * N) .
eq ME * errPNat = errPNat .
eq errPNat * NE = errPNat .

op sd : PNat PNat -> PNat .
op sd : PNat&Err PNat&Err -> PNat&Err .
eq sd(0,N) = N .
eq sd(s(M),0) = s(M) .
eq sd(s(M),s(N)) = sd(M,N) .
eq sd(ME,errPNat) = errPNat .
eq sd(errPNat,NE) = errPNat .

Arithmetic Calculator

```
op _quo_ : PNat PZero -> ErrPNat .
op _quo_ : PNat NzPNat -> PNat .
op _quo_ : PNat&Err PNat&Err -> PNat&Err .
eq M quo 0 = errPNat .
eq M quo s(N) = if M < s(N) then {0} else {s(sd(M,s(N)) quo s(N))} .
eq ME quo errPNat = errPNat .
eq errPNat quo NE = errPNat .

op _rem_ : PNat PZero -> ErrPNat .
op _rem_ : PNat NzPNat -> PNat .
op _rem_ : PNat&Err PNat&Err -> PNat&Err .
eq M rem 0 = errPNat .
eq M rem s(N) = if M < s(N) then {M} else {sd(M,s(N)) rem s(N)} .
eq ME rem errPNat = errPNat .
eq errPNat rem NE = errPNat .   }
```


Arithmetic Calculator

Expressions are defined as follows:

```
mod! EXP {
  [ExpPNat < Exp]
  op 0 : -> ExpPNat {constr} .
  op s : ExpPNat -> ExpPNat {constr }
  op _+_ : Exp Exp -> Exp {constr l-assoc prec: 30} .
  op _-_ : Exp Exp -> Exp {constr l-assoc prec: 30} .
  op _*_ : Exp Exp -> Exp {constr l-assoc prec: 29} .
  op _/_ : Exp Exp -> Exp {constr l-assoc prec: 29} .
  op _%_ : Exp Exp -> Exp {constr l-assoc prec: 29} .
}
```

Arithmetic Calculator

The interpreter is defined as follows:

```
mod! INTER {  pr(PNAT)  pr(EXP)
  op inter : ExpPNat -> PNat .
  op inter : Exp -> PNat&Err .
  var N : PNat .  var EN : ExpPNat .  vars E E1 E2 : Exp .
  eq inter(0) = 0 .
  eq inter(s(EN)) = s(inter(EN)) .
  eq inter(E1 + E2) = inter(E1) + inter(E2) .
  eq inter(E1 - E2) = sd(inter(E1),inter(E2)) .
  eq inter(E1 * E2) = inter(E1) * inter(E2) .
  eq inter(E1 / E2) = inter(E1) quo inter(E2) .
  eq inter(E1 % E2) = inter(E1) rem inter(E2) .
}
```


Arithmetic Calculator

The instructions owned by the virtual machine are as follows:

```
mod! INSTR principal-sort Instr {
  pr(PNAT)
  [Instr]
  op push : PNat -> Instr {constr} .
  op add : -> Instr {constr} .
  op minus : -> Instr {constr} .
  op mult : -> Instr {constr} .
  op div : -> Instr {constr} .
  op mod : -> Instr {constr} .
}
```

Arithmetic Calculator

Generic lists, lists of instructions and stacks of natural numbers are specified as follows:

```
mod! LIST (E :: TRIV) {
  [List]
  op nil : -> List {constr}
  op _|_ : Elt.E List -> List {constr} .
  op _@_ : List List -> List {assoc} .
  var E : Elt.E .
  vars L1 L2 : List .
  -- _@_
  eq nil @ L2 = L2 .
  eq (E | L1) @ L2 = E | (L1 @ L2) .
}
```

```
mod! ILIST {
  pr(LIST(INSTR)
     * {sort List -> IList})
}

mod! STACK {
  pr(LIST(PNAT)
     * {sort List -> Stack,
        op nil -> empstk})
}
```


Arithmetic Calculator

The virtual machine is specified as follows:

```
mod! VM {
  pr(ILIST)
  pr(STACK)
  op vm : IList -> PNat&Err .
  op exec : IList Stack -> PNat&Err .
  var IL : IList .
  var PC : PNat .
  var Stk : Stack .
  var N : PNat .
  vars NE NE1 NE2 : PNat&Err .
```

Arithmetic Calculator

```
eq vm(IL) = exec(IL,empstk) .

eq exec(nil,empstk) = errPNat .
eq exec(nil,NE | empstk) = NE .
eq exec(nil,NE | NE1 | Stk) = errPNat .

eq exec(push(N) | IL,Stk) = exec(IL,N | Stk) .

eq exec(add | IL,empstk) = errPNat .
eq exec(add | IL,NE | empstk) = errPNat .
eq exec(add | IL,NE2 | NE1 | Stk) = exec(IL,NE1 + NE2 | Stk) .

eq exec(minus | IL,empstk) = errPNat .
eq exec(minus | IL,NE | empstk) = errPNat .
eq exec(minus | IL,NE2 | NE1 | Stk) = exec(IL,sd(NE1,NE2) | Stk) .
```


Arithmetic Calculator

```
eq exec(mult | IL,empstk) = errPNat .
eq exec(mult | IL,NE | empstk) = errPNat .
eq exec(mult | IL,NE2 | NE1 | Stk) = exec(IL,NE1 * NE2 | Stk) .

eq exec(div | IL,empstk) = errPNat .
eq exec(div | IL,NE | empstk) = errPNat .
eq exec(div | IL,NE2 | NE1 | Stk) = exec(IL,NE1 quo NE2 | Stk) .

eq exec(mod | IL,empstk) = errPNat .
eq exec(mod | IL,NE | empstk) = errPNat .
eq exec(mod | IL,NE2 | NE1 | Stk) = exec(IL,NE1 rem NE2 | Stk) .
}
```

Arithmetic Calculator

The compiler is specified as follows:

```
mod! COMP { pr(EXP) pr(ILIST)
  op comp : Exp -> IList .
  op en2n : ExpPNat -> PNat .
  var EN : ExpPNat . vars E E1 E2 : Exp .
  eq comp(EN) = push(en2n(EN)) | nil .
  eq comp(E1 + E2) = comp(E1) @ comp(E2) @ (add | nil) .
  eq comp(E1 - E2) = comp(E1) @ comp(E2) @ (minus | nil) .
  eq comp(E1 * E2) = comp(E1) @ comp(E2) @ (mult | nil) .
  eq comp(E1 / E2) = comp(E1) @ comp(E2) @ (div | nil) .
  eq comp(E1 % E2) = comp(E1) @ comp(E2) @ (mod | nil) .
  eq en2n(0) = 0 .
  eq en2n(s(EN)) = s(en2n(EN)) .
}
```


Verification of Arithmetic Calculator Compiler

The properties to be verified are specified:

```
mod! VERIFY-COMP { pr(INTER) pr(VM) pr(COMP)
  op th1 : Exp -> Bool .
  op lem1 : ExpPNat -> Bool .
  op lem2 : Exp IList Stack -> Bool .
  var E : Exp . var EN : ExpPNat . var L : IList . var S : Stack .
  eq th1(E) = (inter(E) = vm(comp(E))) .
  eq lem1(EN) = (inter(EN) = vm(comp(EN))) .
  eq lem2(E,L,S) = (exec(comp(E) @ L,S) = exec(L,vm(comp(E)) | S)) .
}
```

Verification of Arithmetic Calculator Compiler

Let $l(\mathrm{E})$ and $r(\mathrm{E})$ be terms of a same sort in which a variable E of Exp occurs. Then, the following (A) and (B) are equivalent:

(A) $(\forall \mathrm{E}:\mathrm{Exp})\ l(\mathrm{E}) = r(\mathrm{E})$

(B) I. $l(0) = r(0)$ Base case

Induction hypotheses

II. If $l(\mathrm{e}_1) = r(\mathrm{e}_1)$ and $l(\mathrm{e}_2) = r(\mathrm{e}_2)$, Induction case

then $l(\mathrm{e}_1 + \mathrm{e}_2) = r(\mathrm{e}_1 + \mathrm{e}_2)$, $l(\mathrm{e}_1 - \mathrm{e}_2) = r(\mathrm{e}_1 - \mathrm{e}_2)$,

$l(\mathrm{e}_1 * \mathrm{e}_2) = r(\mathrm{e}_1 * \mathrm{e}_2)$, $l(\mathrm{e}_1 / \mathrm{e}_2) = r(\mathrm{e}_1 / \mathrm{e}_2)$ and

$l(\mathrm{e}_1 \% \mathrm{e}_2) = r(\mathrm{e}_1 \% \mathrm{e}_2)$,

where e_1 and e_2 are a fresh constant of Exp.

To prove (A), it suffices to prove (B), which is called proof by *structural induction on Exp*.

Verification of Arithmetic Calculator Compiler

<u>**Theorem 1**</u> [(relative) correctness of the compiler with respect to the interpreter] inter(N) = vm(comp(E))

<u>Proof of Theorem 1</u> By structural induction on E.

I. Base case

```
open VERIFY-COMP .
  -- fresh constants
  op en : -> ExpPNat .
  -- check
  red th1(en) .
close
```

For this proof score fragment, CafeOBJ returns the following:

inter(en) = en2n(en)

which allows us to conjecture the lemma:

```
op lem1 : ExpPNat -> Bool .
eq lem1(EN) = (inter(EN) = vm(comp(EN))) .
```

Verification of Arithmetic Calculator Compiler

Then, the proof score fragment is revised as follows:

```
open VERIFY-COMP .
  -- fresh constants
  op en : -> ExpPNat .
  -- lemmas
  eq inter(EN) = vm(comp(EN)) .
  -- check
  red th1(en) .
close
```


Verification of Arithmetic Calculator Compiler

II. Induction case

```
open VERIFY-COMP .
  -- fresh constants
  ops e1 e2 : -> Exp .
  -- induction hypothesis
  eq inter(e1) = vm(comp(e1)) .
  eq inter(e2) = vm(comp(e2)) .
-- check
  red th1(e1 + e2) .
close
```

For this proof score fragment, CafeOBJ returns the following:

```
exec(comp(e1),empstk) + exec(comp(e2),empstk)
                = exec(comp(e1) @ comp(e2) @ add | nil,empstk)
```

Verification of Arithmetic Calculator Compiler

Let us consider how the following term is likely to be rewritten:

exec(comp(e1) @ comp(e2) @ add | nil,empstk)

$\to^*$ exec(comp(e2) @ add | nil,vm(comp(e1)) | empstk)

$\to^*$ exec(add | nil, vm(comp(e2)) | vm(comp(e1)) | empstk)

$\to$ exec(nil, vm(comp(e1)) + vm(comp(e2)) | empstk)

$\to$ vm(comp(e1)) + vm(comp(e2))

$\to^*$ exec(comp(e1),empstk) + exec(comp(e2),empstk)

Verification of Arithmetic Calculator Compiler

If we have the following equation

exec(comp(E) @ L,S) = exec(L,vm(comp(E)) | S)

then we make it possible to actually conduct the rewriting on the last page. Therefore, we conjecture the lemma:

```
op lem2 : Exp IList Stack -> Bool .
eq lem2(E,L,S)
   = (exec(comp(E) @ L,S) = exec(L,vm(comp(E)) | S)) .
```

Verification of Arithmetic Calculator Compiler

The proof score fragment can be revised as follows:

```
open VERIFY-COMP .
  -- fresh constants
  ops e1 e2 : -> Exp .
  -- lemmas
  eq exec(comp(E) @ L,S) = exec(L,vm(comp(E)) | S) .
  -- induction hypothesis
  eq inter(e1) = vm(comp(e1)) .
  eq inter(e2) = vm(comp(e2)) .
-- check
  red th1(e1 + e2) .
close
```


Verification of Arithmetic Calculator Compiler

The remaining part of the induction case can be tackled likewise.

We need to prove the two lemmas to complete the verification:

Lemma 1 inter(EN) = vm(comp(EN))

Lemma 2 exec(comp(E) @ L,S) = exec(L,vm(comp(E)) | S)

Lemma 1 can be proved by structural induction on EN that is ExpPNat, which does not need any lemmas. Proof by structural induction on ExpPNat is quite similar to proof by structural induction on PNat.

Lemma 2 can be proved by structural induction on E that is Exp, which does not need any lemmas.

Exercises

1. Complete the verification.